

An Efficient Binary Technique for Trace Simplifications of Concurrent Programs

Mohamed A. El-Zawawy*

Mohammad N. Alanazi†

*†College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia

*Department of Mathematics, Faculty of Science
Cairo University
Giza 12613, Egypt

Email*: maelzawawy@cu.edu.eg
Email†: alanazi@ccis.imamu.edu.sa

Abstract—Execution of concurrent programs implies frequent switching between different thread contexts. This property perplexes analyzing and reasoning about concurrent programs. Trace simplification is a technique that aims at alleviating this problem via transforming a concurrent program trace (execution) into a semantically equivalent one. The resulted trace typically includes less number of context switches than that in the original trace.

This paper presents a new static approach for trace simplification. This approach is based on a connectivity analysis that calculates for each trace-point connectivity and context-switching information. The paper also presents a novel operational semantics for concurrent programs. The semantics is used to prove the correctness and efficiency of the proposed techniques for connectivity analysis and trace simplification. The results of experiments testing the proposed technique on problems treated by previous work for trace simplification are also shown in the paper. The results prove the efficiency and effectiveness of the proposed method.

I. INTRODUCTION

Concurrency [15] is becoming a main stream in programming due to advances in multi-core hardware. Compared to other programming techniques, debugging and reasoning about concurrent programs are not easy jobs; in fact they are very difficult. This is mainly because of the non-deterministic behavior of their executions. The debugging difficulty was reported by research [6] comparing debugging resources needed for concurrent and sequential programs where debugging the former was found to last, on average, (more than twice) longer than debugging the latter. The non-deterministic behavior of execution is caused by non-deterministic thread interleaving at execution time. This makes reproducing a bug towards analyzing and resolving it, in most cases, difficult. Much research [14] has been carried out for smoothing bug reproductions in concurrent programs.

Context switching [15] is a terminology describing (fine-grained) interleaving of different threads. A relatively large number of context switches in an execution of a concurrent program complicates its debugging extremely. This is so as the number of possible interactions between threads needing to be reasoned about, in order to understand a trace (an execution), becomes extremely huge by thread interleaving. Therefore, it is quite helpful to produce an equivalent execution trace (of

a given one) that has less number of context switches. This results in increasing the interleaving granularity. One main source of increase in context switches is thinking sequentially while coding concurrently. Few attempts [10] were done to produce techniques for reduction of context switches in executions (traces) of concurrent programs.

This paper presents a new technique, Binary Trace Reduction (*BinTrcRed*), for automatic reductions of context switches in traces (execution instances) of concurrent programs. This technique (transformation) produces an equivalent trace to the given one and hence the produced trace maintains bugs of the original one. Therefore the resulted simplified trace can be useful in the debugging process as it removes the burden of reasoning about unnecessary fine-grained thread interactions. The proposed technique has the form of a system of inference rules. This has two advantages over related work. First the system is relatively easy to understand and to apply as it is simply structured. Secondly, the system naturally associates each trace simplification process with a validity proof which has the form a rule derivation in the system. This proof is required by many applications like proof-carrying code [12].

BinTrcRed is based on the result of a connectivity analysis that is proposed in this paper and that also has the form of a system of inference rules. The connectivity analysis simply analyzes a given trace towards complete information about the number of context switches and trace-joins where switching takes place. Then based on this information, a sequence of *binary* replacements between segments (sequence) of statements constituting the trace are performed by *BinTrcRed* to reduce the number of context switches. *BinTrcRed* computes a locally optimal simplification rather than a globally optimal simplification as the problem was proved to be NP-hard [11].

Two measures are used to verify the correctness and efficiency of the proposed technique. The first measure is theoretical and provides a robust ground for *BinTrcRed*. This is done via designing an accurate, yet simple, operational semantics for the model language used in this paper. This model is used to state and prove the correctness and efficiency of *BinTrcRed*. More specifically, the semantics is used to prove that any resulting trace by *BinTrcRed* is equivalent (having the same effect on memory) to the input one and has a number

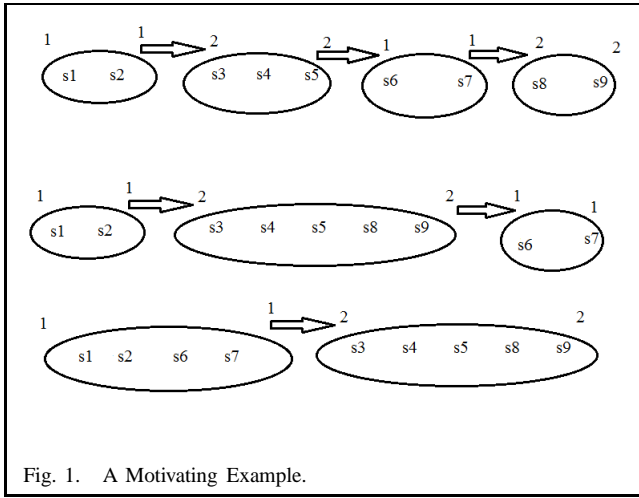


Fig. 1. A Motivating Example.

of context switches that is less than or equal to that in the original trace. The other measure is experiential results that were carried out to compare the performance of *BinTrcRed* to a previous technique. Many parameters were used towards a fair comparison. Experiments confirm that our technique is faster and more effective than the previous technique.

Figure 1 presents a motivating example of the work proposed in this paper. Assume a concurrent program P that includes 9 statements distributed between 2 threads. The upper part of Figure 1 presents a trace of executing this program. This trace includes 4 groups of connected statements and 3 context switches. For example statements $s3$, $s4$ and $s5$ are connected and included in thread 2. After $s5$, a context switch happens to thread 1 to execute the connected statements $s6$ and $s7$. First of all, a robust analysis to accurately collect such connectivity and switching information is required. Base of the connectivity information if we replace the third and fourth groups of statements we get the equivalent trace at the middle of the figure with 2 context switches. Intuitively, the equivalency is due to the replacement of unconnected groups of statements. Further replacements produce the final equivalent trace at the bottom of the figure with only 1 context switch. This paper aims at formalizing a technique that does such replacements. The technique is required also to associate each such transformation with a correctness proof that is compact enough for the sake of mobility.

Contributions of this paper are the following:

- 1) A new operational approach to accurately define the semantics of concurrent programs.
- 2) A connectivity analysis to calculate connectivity and context switching information in traces of concurrent programs.
- 3) A new technique to reduce context switches in traces of concurrent programs.

The outline of this paper is as follows. Section II presents the used model of programming language and the proposed techniques for connectivity analysis and trace transformation. The semantics for the language constructs together with a formalization for correctness and efficiency of proposed techniques are shown in Section III. Section IV presents the

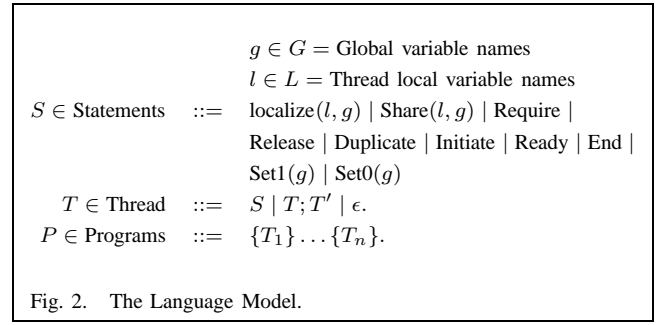


Fig. 2. The Language Model.

experimental results. Related and future research are reviewed in Section V.

II. CONNECTIVITY ANALYSIS AND TRACE TRANSFORMATION

This section presents our model for a concurrent programming language. The section also presents two techniques; a connectivity analysis and a trace transformation reducing number of context switches in concurrent programs (trace simplification). The section also uses the language model to introduce a formalization of the problem of trace simplification (*BinTrcRed*). The language model includes commands common to languages used to study similar problems. Figure 2 presents the language model.

Some comments on the language model are in order. Two types of stores are used in the model; global (typical element denoted by g) and local (typical element denoted by l). A global store is a memory location that is accessed by all threads constituting a concurrent program. A local store is a memory location that is private for a certain thread. A special global store, tc , serves as a counter for the trace. Global stores are meant to facilitate the communication among a program threads. According to the syntax of Figure 2, each thread consists of a sequence of statements. Due to the use of a global trace counter, standing alone, each thread is deterministic.

Towards a rich, yet simple, language model the syntax of our language includes the following statements:

- *Share*(l, g): copying the value of g into l . (Java's read command)
- *localize*(l, g): copying the value of l to g . (Java's write command)
- *Require*: meaning that its hosting thread requires a lock. (Java's lock command)
- *Release*: meaning that its hosting thread releases a lock. (Java's unlock command)
- *Duplicate*: meaning that its hosting thread duplicates itself. (Java's fork command)
- *Initiate*: meaning that the execution of its hosting thread is initiated immediately after the completion of another thread. (Java's join command)
- *Ready*: meaning that its hosting thread is ready for execution. (Java's start command)

- *End*: marking the end of a thread. (Java's exit command)
- *Set1(g)*: setting the value of g to 1. (Java's signal command)
- *Set0(g)*: waiting g to become 1 to set it to 0 again. (Java's wait command)

Definition 1: Let $P = \{T_1\} \dots \{T_n\}$ be a program and suppose that T_i has n_i statements (i.e. $T_i = S_1^i; \dots; S_{n_i}^i$). Then

- 1) $N_P = \sum_i n_i$.
- 2) $S_P = \{S_j^i \mid 1 \leq i \leq n \ \& \ 1 \leq j \leq n_i\}$
- 3) A faithful map δ_P for the program P is a one-to-one map

$$\delta_P : \{1, \dots, N_P\} \rightarrow S_P$$

satisfying the following condition:

$u, v \in \{1, \dots, N_P\}$, $\delta_P(u) = S_{q_1}^i$, and $\delta_P(v) = S_{q_2}^i \implies q_1 < q_2$.

- 4) The trace, t_{δ_P} , of δ_P is the sequence $\delta_P(1); \delta_P(2); \dots; \delta_P(N_P)$.
- 5) For $u \in \{1, \dots, N_P\}$, suppose $\delta_P(u) = S_{q_1}^i$. Then

$$th_{\delta_P} : \{1, \dots, N_P\} \rightarrow \{1, \dots, n\}; s \mapsto i.$$

- 6) For $u, v \in \{1, \dots, N_P\}$,

$$\text{diff}(u, v) = \begin{cases} 0, & th_{\delta_P}(u) = th_{\delta_P}(v); \\ 1, & \text{otherwise.} \end{cases}$$

- 7) $CS(t_{\delta_P}) = \sum_{s=1}^{N_P-1} \text{diff}(s, s+1)$.

Definition 1 introduces concepts necessary to introduce results of the paper and to formalize the problem of trace simplification. Some comments on the definition above are in order. The number and the set of all statements in all threads of a concurrent program P are denoted by N_P and S_P , respectively. We assume that each program statement is superscribed with its thread number. Hence when necessary, a trace is denoted by $S_1^{i_1}, \dots, S_{N_P}^{i_{N_P}}$, where $1 \leq i_1, \dots, i_{N_P} \leq n$. A faithful map (denoted by δ_P) of a concurrent program P , is a map that orders the program statements in way that respects the inner order of each thread. Hence each trace (denoted by t_{δ_P}) can be realized as the image of a faithful map δ_P . For a faithful map $\delta_P(u)$, the map th_{δ_P} calculates for a given position in the trace, the ID of the thread that hosts the statement occupying the position. Using the map th_{δ_P} of a faithful map $\delta_P(u)$, the map $\text{diff}(u, v)$ decides whether locations number u and v of the trace are hosting statements of the same thread. Therefore the summation $\sum_{s=1}^{N_P-1} \text{diff}(s, s+1)$ is the number of the context switches of the trace in hand.

Definition 2: For a trace $S_1^{i_1}, \dots, S_{N_P}^{i_{N_P}}$ of a program $P = \{T_1\} \dots \{T_n\}$,

- $C_1^P = \{(S_1, S_2) \mid \exists 1 \leq i \leq n, 1 \leq j \leq n_i. S_1 = S_j^i \text{ and } S_2 = S_{j+1}^i\}$.
- $C_2^P = \{(\text{Release}^i, \text{Require}^i), (\text{Duplicate}^i, \text{Read}^i), (\text{End}^i, \text{Initiate}^i), (\text{Set1}^i(g), \text{Set0}^{i'}(g)) \mid 1 \leq i, i' \leq n\}$.
- $C_3^P = \{(\text{localize}^i(l, g), \text{Share}^j(l', g)), (\text{Share}^i(l, g), \text{localize}^j(l', g)), (\text{Share}^i(l, g), \text{Share}^j(l', g)) \mid i \neq j\}$.

- The connectivity set:

$$C^P = C_1^P \cup C_2^P \cup C_3^P.$$

- The map, connect, measuring connectivity of statements in a trace is defined as following:

$$\text{connect}(S_1, S_2) = \begin{cases} 1, & (S_1, S_2) \in C^P; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 3: For a trace S_1, \dots, S_{N_P} of a program $P = \{T_1\} \dots \{T_n\}$, an annotated trace is a sequence:

$$(s_1^0, s_2^0, t_1^0, t_2^0) S_1 (s_1^1, s_2^1, t_1^1, t_2^1) S_2 (s_1^2, s_2^2, t_1^2, t_2^2) \dots S_{N_P} (s_1^{N_P}, s_2^{N_P}, t_1^{N_P}, t_2^{N_P}),$$

such that $\forall u \in \{1, \dots, N_P\}$, $s_1^u, s_2^u \in \{1, \dots, N_P\}$ and $t_1^u, t_2^u \in \{1, \dots, n\}$.

The conditions under which two statements of a concurrent program are considered connected are presented in Definition 2. There are three types of pairs of connected statements in a trace of a program P . The three types are the following. Pairs of contiguous statements of the same thread are grouped in the set C_1^P . The set C_2^P collects pairs of contiguous concurrent statements of various threads accessing the same global variable. Pairs of contiguous conflicting concurrent statements are grouped in the set C_3^P . The set of all pairs of connected statements is denoted by C^P . The map $\text{connect}(S_1, S_2)$ is binary-valued and decides connectivity of S_1 and S_2 using the set C^P . To illustrate Definition 3, it is necessary to recall that each trace consists of contiguous segments of statements such that inside each segment contiguous statements are connected. In an extreme case, each segment includes only one statement. For a trace, Definition 3 introduces the concept of an annotated trace which is a trace whose join-points are annotated with connectivity information. For a join-point i , this information is a quadruple $(s_1^i, s_2^i, t_1^i, t_2^i)$ where:

- the number of the first member in the segment including the statement S_i is denoted by s_1^i ,
- the number of the last member in the segment including the statement S_i is denoted by s_2^i ,
- the thread ID of the first member in the segment including the statement S_i is denoted by t_1^i , and
- the thread ID of the last member in the segment including the statement S_i is denoted by t_2^i .

Figure 3 presents the connectivity analysis in the form of a system of inference rules. For a given trace S_1, \dots, S_{N_P} of a program $P = \{T_1\} \dots \{T_n\}$, the idea is to use the rules to find a quadruple (s_1, s_2, t_1, t_2) such that

$$S_1, S_2, \dots, S_{N_P} : (0, 0, 0, 0) \rightarrow (s_1, s_2, t_1, t_2)$$

is derivable in the system. If such derivation exists, then an annotated trace (in the sense of Definition 3 above) can be easily built from the derivation. The obtained annotated trace includes all the necessary connectivity information for embarking on reducing the number of context switches. The precondition of (tr_3) , $\text{connect}(\delta_P(u-1), \delta_P(u)) = 1$ requires that the current statement is connected to its prior one. In this case the current statement is attached to the segment of its prior statement by letting the information in the next join-point to be $(s_1, u, t_1, th_{\delta_P}(u))$.

$$\begin{array}{c}
\frac{}{S_u : (0, 0, 0, 0) \rightarrow (u, u, th_{\delta_P}(u), th_{\delta_P}(u))} \text{(tr}_1\text{)} \\
\frac{\text{connect}(\delta_P(u-1), \delta_P(u)) = 0}{S_u : (s_1, s_2, t_1, t_2) \rightarrow (u, u, th_{\delta_P}(u), th_{\delta_P}(u))} \text{(tr}_2\text{)} \\
\frac{\text{connect}(\delta_P(u-1), \delta_P(u)) = 1}{S_u : (s_1, s_2, t_1, t_2) \rightarrow (s_1, u, t_1, th_{\delta_P}(u))} \text{(tr}_3\text{)} \\
\frac{S_1 : (s_1, s_2, t_1, t_2) \rightarrow (s'_1, s'_2, t'_1, t'_2) \quad S_2, \dots, S_q : (s'_1, s'_2, t'_1, t'_2) \rightarrow (s'_1, s'_2, t'_1, t'_2)}{S_1, S_2, \dots, S_q : (s_1, s_2, t_1, t_2) \rightarrow (s'_1, s'_2, t'_1, t'_2)} \text{(tr}_4\text{)}
\end{array}$$

Fig. 3. Rules for Connectivity Analysis.

$$\begin{array}{c}
\frac{(s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u)}{\Rightarrow (s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u)} \text{(base}_0\text{)} \\
\frac{t_2^{u-1} \neq t_1^{u+1}}{(s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u) S_{u+1} \Rightarrow (s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u) S_{u+1} (s_1^{u+1}, s_2^{u+1}, t_1^{u+1}, t_2^{u+1})} \text{(base}_1\text{)} \\
\frac{t_2^{u-1} = t_1^{u+1}}{(s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u) S_{u+1} \Rightarrow (s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u) S_{u+1} (s_1^{u+1}, s_2^{u+1}, t_1^{u+1}, t_2^{u+1})} \text{(base}_2\text{)} \\
\frac{N_q > 2 \quad v = \lceil \mu/2 \rceil \quad (s_1^0, s_2^0, t_1^0, t_2^0) S_1 (s_1^1, s_2^1, t_1^1, t_2^1) S_2 \dots S_v (s_1^v, s_2^v, t_1^v, t_2^v) \Rightarrow (s_1^{0'}, s_2^{0'}, t_1^{0'}, t_2^{0'}) S'_1 (s_1^{1'}, s_2^{1'}, t_1^{1'}, t_2^{1'}) S'_2 \dots S'_v (s_1^{v'}, s_2^{v'}, t_1^{v'}, t_2^{v'}) S_{v+1} (s_1^{v+1}, s_2^{v+1}, t_1^{v+1}, t_2^{v+1}) S_{v+2} \dots S_\mu (s_1^\mu, s_2^\mu, t_1^\mu, t_2^\mu) \Rightarrow (s_1^{0'}, s_2^{0'}, t_1^{0'}, t_2^{0'}) S'_1 (s_1^{1'}, s_2^{1'}, t_1^{1'}, t_2^{1'}) S'_2 \dots S'_v (s_1^{v'}, s_2^{v'}, t_1^{v'}, t_2^{v'}) S_{v+1} (s_1^{v+1}, s_2^{v+1}, t_1^{v+1}, t_2^{v+1}) S_{v+2} \dots S'_\mu (s_1^\mu, s_2^\mu, t_1^\mu, t_2^\mu)} \text{(S)} \\
\Rightarrow \begin{cases} (s_1^{0'}, s_2^{0'}, t_1^{0'}, t_2^{0'}) S'_1 (s_1^{1'}, s_2^{1'}, t_1^{1'}, t_2^{1'}) S'_2 \dots S'_v (s_1^{v'}, s_2^{v'}, t_1^{v'}, t_2^{v'}) S_{v+1} (s_1^{v+1}, s_2^{v+1}, t_1^{v+1}, t_2^{v+1}) S_{v+2} \dots S'_\mu (s_1^\mu, s_2^\mu, t_1^\mu, t_2^\mu) & \text{if } t_2^0 = t_1^\mu; \\ (s_1^0, s_2^0, t_1^0, t_2^0) S_1 (s_1^1, s_2^1, t_1^1, t_2^1) S_2 \dots S_\mu (s_1^\mu, s_2^\mu, t_1^\mu, t_2^\mu), & \text{otherwise.} \end{cases}
\end{array}$$

Fig. 4. *BinTrcRed*: Rules for Context Switching Reduction.

It is quite important to note that although the proposed connectivity analysis seems to cost $O(2^n)$, this is not the case as the method does not actually ensure the connectivity for all pairs of statements. However, the proposed method ensures connectivity of (roughly) all join points of the input program.

Figure 4 presents *BinTrcRed*, the main technique of the paper for reducing number of context switches in concurrent program. The technique has the form of a system of inference rules. The technique builds on the results of the connectivity analysis introduced above. The rule (base₀) expresses the fact that the transformation of a single statement is the statement itself again. For the annotated trace

$$(s_1^{u-1}, s_2^{u-1}, t_1^{u-1}, t_2^{u-1}) S_u (s_1^u, s_2^u, t_1^u, t_2^u) S_{u+1} (s_1^{u+1}, s_2^{u+1}, t_1^{u+1}, t_2^{u+1}),$$

if $t_2^{u-1} \neq t_1^{u+1}$, then switching the two statements S_u and S_{u+1} would not reduce the number of context switches. Therefore as formalized in the rule (base₁), the transformation of the trace above is the same trace again. However if $t_2^{u-1} = t_1^{u+1}$, then switching the two statements S_u and S_{u+1} would reduce the number of context switches in the trace by one. This is formalized in the rule (base₂). For a longer annotated trace, the rule (S) breaks the trace into two sub-traces and applies the system on each sub-trace. Then the rule switches the two obtained sub-traces only if their switching would reduce the number of context switches by 1.

Theorem 1 states that the number of context switching in a trace resulted from the transformation system above, *BinTrcRed*, is less than or equal that number in the ordinal trace. A straightforward structure induction on rules of Figure 4 proves the theorem.

Theorem 1: Let $P = \{T_1\} \dots \{T_n\}$ be a program and suppose that T_i has n_i statements (i.e. $T_i = S_1^i; \dots; S_{n_i}^i$). Suppose that δ_P is a trace for P with annotation:

$$(s_1^0, s_2^0, t_1^0, t_2^0) \delta_P(1) (s_1^1, s_2^1, t_1^1, t_2^1) \delta_P(2) (s_1^2, s_2^2, t_1^2, t_2^2) \dots \delta_P(N_P) (s_1^{N_P}, s_2^{N_P}, t_1^{N_P}, t_2^{N_P}),$$

obtained using the analysis technique of Figure 3. Suppose that this trace is transomed using *BinTrcRed* (Figure 4):

$$(s_1^0, s_2^0, t_1^0, t_2^0) \delta_P(1) (s_1^1, s_2^1, t_1^1, t_2^1) \dots \delta_P(N_P) (s_1^{N_P}, s_2^{N_P}, t_1^{N_P}, t_2^{N_P}) \Rightarrow (s_1^{0'}, s_2^{0'}, t_1^{0'}, t_2^{0'}) \delta'_P(1) (s_1^{1'}, s_2^{1'}, t_1^{1'}, t_2^{1'}) \dots \delta'_P(N_P) (s_1^{N_P'}, s_2^{N_P'}, t_1^{N_P'}, t_2^{N_P'}),$$

Then $CS(\delta'_P) \leq CS(\delta_P)$.

III. SEMANTICS BASED CORRECTNESS FORMALIZATION

This section presents a novel semantics for trace executions in concurrent programming languages. The proposed semantics is operational and consists of a set of states and a transition relation between the states. A state is a triple (γ, L, W) , where γ captures the contents of local and global variables, L is the set of threads requiring looks at that point of execution, and W is the set of global variables being watched by the command Set0. Definition 4 formalizes the state definition.

Definition 4: • Local locations of thread i are $L_i = \{l_1^i, l_2^i, \dots\}$.

- A special global variable is the trace counter denoted by tc .
- A variable state γ is a partial map from $G \cup \cup_i L_i$ to the set of integers.
- A trace state is a triple (γ, L, W) ; L denotes a list of threads requiring a lock and W denotes the set of global variables being watched by the statement "Set0".

The transition relation of the proposed operational semantics, in the form of a system of inference rules, is shown in Figure 5. Some comments are in order. The rule (f^s) simulates the semantics of the statement Duplicate ^{i} . This is done via adding statements of thread i into the set S_P of all statements of the program P after removing the already executed statements from S_P . The remaining trace is replaced

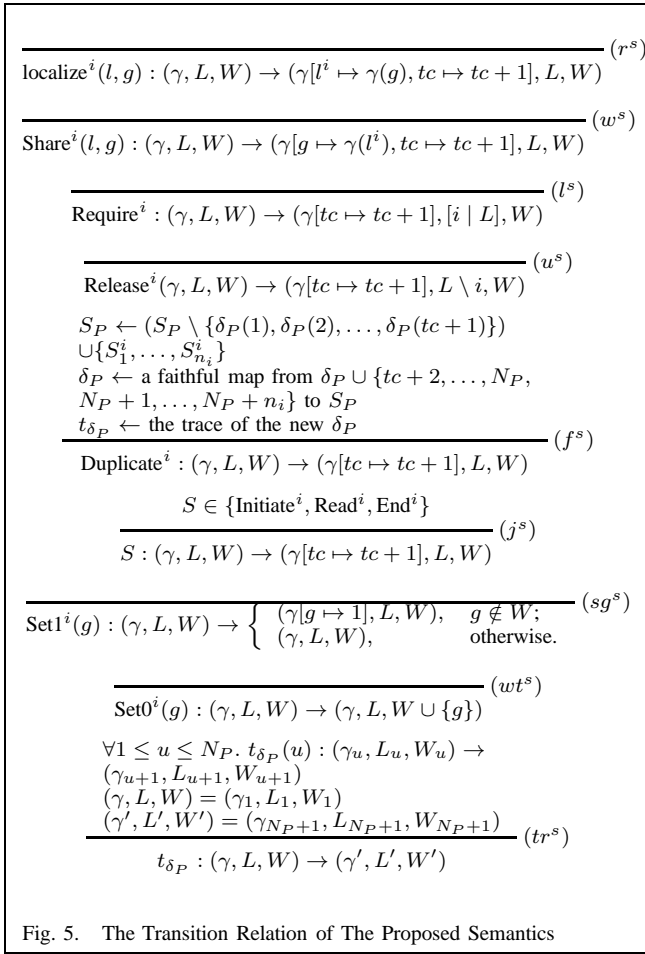


Fig. 5. The Transition Relation of The Proposed Semantics

with the new trace corresponding to a faithful map for the new set of all statements.

Theorem 2 formalizes the correctness of the transformation technique, *BinTrcRed*, proposed in the previous section. This is done using the operational semantics detailed above. The proof of the theorem is built using structure induction on transformation and semantics rules.

Theorem 2: Let $P = \{T_1\} \dots \{T_n\}$ be a program and suppose that T_i has n_i statements (i.e. $T_i = S_1^i; \dots; S_{n_i}^i$). Suppose that δ_P is a trace for P with annotation:

$$(s_1^0, s_2^0, t_1^0, t_2^0) \delta_P(1) (s_1^1, s_2^1, t_1^1, t_2^1) \delta_P(2) s_1^2, s_2^2, t_1^2, t_2^2) \dots (s_1^{N_P}, s_2^{N_P}, t_1^{N_P}, t_2^{N_P}),$$

obtained using the analysis technique of Figure 3. Suppose that this trace is transduced using *BinTrcRed*:

$$(s_1^0, s_2^0, t_1^0, t_2^0) \delta_P(1) (s_1^1, s_2^1, t_1^1, t_2^1) \dots \delta_P(N_P) (s_1^{N_P}, s_2^{N_P}, t_1^{N_P}, t_2^{N_P}) \implies (s_1^{0'}, s_2^{0'}, t_1^{0'}, t_2^{0'}) \delta'_P(1) (s_1^{1'}, s_2^{1'}, t_1^{1'}, t_2^{1'}) \dots \delta'_P(N_P) (s_1^{N_P'}, s_2^{N_P'}, t_1^{N_P'}, t_2^{N_P'}).$$

Suppose that for some (γ, L, W) ,

$$t_{\delta_P} : (\gamma, L, W) \rightarrow (\gamma', L', W')$$

and

$$t_{\delta'_P} : (\gamma, L, W) \rightarrow (\gamma'', L'', W'').$$

Then

$$(\gamma', L', W') = (\gamma'', L'', W'').$$

IV. IMPLEMENTATION AND EVALUATION

In order to investigate the effectiveness and efficiency of *BinTrcRed*, several experiments were performed on an implementation of our proposed technique. *BinTrcRed* was implemented as a prototype tool for multithreaded Java programs. The tool includes six phases. The first phase calculates the number of context switches in the given trace of execution. The second phase applies the connectivity analysis (Figure 3) to annotate each point of the given trace with connectivity information. The third phase calculates the semantics (Figure 5) of the trace. The fourth phase uses the connectivity information and the optimization rules (Figure 4) to reduce the trace. The fifth phase calculates the number of context switches in the resulted trace. The last phase calculates the semantics of the resulted trace. Calculating the number of context switches and semantics before and after transformations makes *BinTrcRed* transparent to the programmers.

Four common multithreaded Java benchmarks were the subject of our experiments. The first benchmark, CTSP, is a multithreaded solution for traveling salesman problem using a concurrent bound and branch algorithm. The second benchmark, CPhilo, simulates the famous dining philosophers problem. The third benchmark, CWebDow, is a multithreaded tool for downloading from servers and servers reflection. The fourth benchmark, CMerge, is a multithreaded version of the merge sort algorithm. The experiments were run on a Windows 7 system whose processor is Intel(R)-Core2(TM)-i5-CPU-(2.53GHz) and whose RAM is 4GB.

The experimental results are shown in Table I. For the sake of accuracy, all information are averaged using results of 100 runs. Parameters used to measure the performance are the following.

- 1) LC: Numbers of lines in source programs.
- 2) TC: Thread counts.
- 3) SR_b : The semantics running-time before transformation.
- 4) CR: Connectivity analysis running-time.
- 5) TR: Trace-transformation running-time.
- 6) SR_a : The semantics running-time after transformation.
- 7) CS_b : The number of context switches before transformation.
- 8) CS_a : The number of context switches after transformation.

The following comments about results worth mentioning. It is noted that the trace-transformation run-time (TR) is proportional to the original number of context switches. The semantics run-time before transformation is typically more than that after transformation. This is justified with the reduction in the number of context switches. The proposed algorithm managed to reduce number of context switches by 85.3% on average. This improves on the result of *SimTrace* [10] whose average reduction percentage is 83.8%. Compared to *SimTrace*, the binary nature of our proposed technique, *BinTrcRed*, makes it more efficient for larger traces. All in all, compared to the state of the art, these results prove the value and usefulness (regarding efficiency and trace simplification) of the proposed techniques. Two important advantages of our proposed technique over related ones is that our technique is supported with

	LC	TC	SR _b	CR	TR	SR _a	CS _b	CS _a
CPhilo	81	6	4.0 ms	5.0 ms	5.0 ms	3.0 ms	54	8
CMerge	519	18	25.0 ms	29.0 ms	32.0 ms	26.0 ms	541	93
CTSP	709	5	68.0 ms	78.0 ms	97.0 ms	54.0 ms	9617	1143
CWebDow	35175	3	43.0 ms	48.0 ms	42.0 ms	33.0 ms	144	21

TABLE I. EXPERIMENTAL RESULTS

the operational semantics and a correctness proof for each trace transformation. The correctness proofs have the form of inference rules derivations. This has many applications; specially in the proof-carrying code area of research.

V. RELATED WORK

Towards finding bug cases in error traces, many algorithms [8], [13] for checking software models have been proposed. Most of these algorithms aim at building counterexamples in case of finding a bug and aim also at reducing error traces. Required changes in thread scheduling to get an error that is concurrency-based was achieved by an extended version of delta debugging [1]. Assuming the existence of rely-guarantee proofs for concerned properties, in [7] concurrent programs were verified. Although the proposed technique in the current paper relies on producing reductions in single traces, the techniques mentioned above rely on comparing related traces. Clearly, focusing on reducing a single trace is more practical and efficient but creates a more complicated scenario.

A static approach, *SimTrace*, to trace simplification is proposed in [10]. The idea behind *SimTrace* is to use dependence graphs to model events. Rather than introducing a trace theorem for equivalence, in [10] it is proved that results of *SimTrace* are sound. Hence in this approach re-execution of program for the sake of validation is not required. The use of a dependence relation [10] is a common practice in treating trace optimizations. Checking violations of atomicity was achieved in [18] via the introduction of concept of guarded independence. To minimize the cardinality of the causality relationship, the concept of sliced causality was introduced in [3]. This was done by shopping the typical dependencies among commands. Other research [17] considered all possible valid executions that may result from a trace. This was done using a model for maximal causality. The rule of dependence relation is achieved in our proposed technique, *BinTrcRed*, by the connectivity analysis which is simpler and more powerful than the mentioned techniques due to simplicity of inference rules as they were explained earlier.

In [16], a theory of context-bounded analysis was developed for concurrent programs. Up to the bound, this theory is both sound and complete. Results concerning sequential pushdown systems [2], in particular their model checking, were used to develop this theory. Many model checkers have been proposed for concurrent programs [4]. The problem with all these checkers is that they use a representation of the stacks of threads. Non-termination may occur due to such stacks. Other techniques [9] for verifying concurrent programs that are automated have also been developed. The idea in these techniques is to use an automatically established model of the environment to separately check each process. This checking model suffers from being imprecise and stackless. Therefore such techniques are not complete, but sound.

REFERENCES

- [1] Cyrille Artho. Iterative delta debugging. *STTT*, 13(3):223–246, 2011.
- [2] Kshitij Bansal and Stéphane Demri. Model-checking bounded multi-pushdown systems. In Andrei A. Bulatov and Arseny M. Shur, editors, *CSR*, volume 7913 of *Lecture Notes in Computer Science*, pages 405–417. Springer, 2013.
- [3] Feng Chen and Grigore Rosu. Parametric and sliced causality. In Damm and Hermanns [5], pages 240–253.
- [4] Chia Yuan Cho, Vijay D’Silva, and Dawn Song. Blitz: Compositional bounded model checking for real-world programs. In *ASE*, pages 136–146. IEEE, 2013.
- [5] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.
- [6] Martin Dimitrov and Huiyang Zhou. Time-ordered event traces: A new debugging primitive for concurrency bugs. In *IPDPS*, pages 311–321. IEEE, 2011.
- [7] Pranav Garg and P. Madhusudan. Compositionality entails sequentializability. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2011.
- [8] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
- [9] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 331–344. ACM, 2011.
- [10] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In Eran Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2011.
- [11] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *SIGSOFT FSE*, pages 57–66. ACM, 2010.
- [12] Romain Jobredeaux, Heber Herencia-Zapana, Natasha A. Neogi, and Eric Feron. Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems. In *CDC*, pages 1816–1821. IEEE, 2012.
- [13] K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.
- [14] Hongwei Liao, Yin Wang, Hyoun Kyu Cho, Jason Stanley, Terence Kelly, Stéphane Lafortune, Scott A. Mahlke, and Spyros A. Reveliotis. Concurrency bugs in multithreaded software: modeling and analysis using petri nets. *Discrete Event Dynamic Systems*, 23(2):157–195, 2013.
- [15] Peter Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. 1 edition (2011).
- [16] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
- [17] Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In Shaz Qadeer and Serdar Tasiran, editors, *RV*, volume 7687 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2012.
- [18] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2010.